

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01\_The\_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02\_Basic\_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03\_Strings\_and\_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04\_Collection\_Types.html)

Via 由 [ 极客学院 Wiki

(http://wiki.jikexueyuan.com) ]

提供

## 构造过程（Initialization）

1.0 翻译: lifedim (https://github.com/lifedim) 校对: lifedim (https://github.com/lifedim)
2.0 翻译+校对: chenmingbiao (https://github.com/chenmingbiao)
2.1 翻译: Channe (https://github.com/Channe), Realank (https://github.com/Realank) 校对: shanks (http://codebuild.me), 2016-1-23
2.2 翻译: pmst (https://github.com/colourful987) 翻译+校对: SketchK (https://github.com/SketchK) 2016-05-14

本页包含内容:

- 存储属性的初始赋值
- 自定义构造过程
- 默认构造器
- 值类型的构造器代理
- 类的继承和构造过程
- 可失败构造器
- 必要构造器
- 通过闭包或函数设置属性的默认值

构造过程是使用类、结构体或枚举类型的实例之前的准备过程。在新实例可用前必须执行这个过程，具体操作包括设置实例中每个存储型属性的初始值和执行其他必须的设置或初始化工作。

通过定义构造器（`Initializers`）来实现构造过程，这些构造器可以看做是用来创建特定类型新实例的特殊方法。与 Objective-C 中的构造器不同，Swift 的构造器无需返回值，它们的主要任务是保证新实例在第一次使用前完成正确的初始化。

类的实例也可以通过定义析构器（`deinitializer`）在实例释放之前执行特定的清除工作。想了解更多关于析构器的内容，请参考析构过程（./15\_Deinitialization.html）。

### 存储属性的初始赋值

类和结构体在创建实例时，必须为所有存储型属性设置合适的初始值。存储型属性的值不能处于一个未知的状态。

你可以在构造器中为存储型属性赋初值，也可以在定义属性时为其设置默认值。以下小节将详细介绍这两种方法。

注意

当你为存储型属性设置默认值或者在构造器中为其赋值时，它们的值是被直接设置的，不会触发任何属性观察者（`property observers`）。

### 构造器

构造器在创建某个特定类型的新实例时被调用。它的最简形式类似于一个不带任何参数的实例方法，以关键字 `init` 命名：

```
init() {
    // 在此处执行构造过程
}
```

下面例子中定义了一个用来保存华氏温度的结构体 `Fahrenheit`，它拥有一个 `Double` 类型的存储型属性 `temperature`：

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// 输出 "The default temperature is 32.0° Fahrenheit"
```

这个结构体定义了一个不带参数的构造器 `init`，并在里面将存储型属性 `temperature` 的值初始化为 `32.0`（华氏温度下水的冰点）。

默认属性值

Wiki > 移动开发 > iOS > The Swift Programming Language 中文版

如前所述，你可以在构造器中为存储型属性设置初始值。同样，你也可以在属性声明时为其设置默认值。

注意

如果一个属性总是使用相同的初始值，那么为其设置一个默认值比每次都在构造器中赋值要好。两种方法的效果是一样的，只不过使用默认值让属性的初始化和声明结合得更紧密。使用默认值能让你的构造器更简洁、更清晰，且能通过默认值自动推导出属性的类型；同时，它也能让你充分利用默认构造器、构造器继承等特性（后续章节将讲到）。

你可以使用更简单的方式在定义结构体 Fahrenheit 时为属性 temperature 设置默认值：

```
struct Fahrenheit {
    var temperature = 32.0
}
```

自定义构造过程

你可以通过输入参数和可选类型的属性来自定义构造过程，也可以在构造过程中修改常量属性。这些都将在后面章节中提到。

构造参数

自定义 构造过程 时，可以在定义中提供构造参数，指定所需值的类型和名字。构造参数的功能和语法跟函数和方法的参数相同。

下面例子中定义了一个包含摄氏温度度的结构体 Celsius 。它定义了两个不同的构造器：init(fromFahrenheit:) 和 init(fromKelvin:) ，二者分别通过接受不同温标下的温度值来创建新的实例：

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}

let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius 是 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius 是 0.0"
```

第一个构造器拥有一个构造参数，其外部名字为 fromFahrenheit ，内部名字为 fahrenheit ；第二个构造器也拥有一个构造参数，其外部名字为 fromKelvin ，内部名字为 kelvin 。这两个构造器都将唯一的参数值转换成摄氏温度值，并保存在属性 temperatureInCelsius 中。

参数的内部名称和外部名称

跟函数和方法参数相同，构造参数也拥有一个在构造器内部使用的参数名字和一个在调用构造器时使用的外部参数名字。

然而，构造器并不像函数和方法那样在括号前有一个可辨别的名字。因此在调用构造器时，主要通过构造器中的参数名和类型来确定应该被调用的构造器。正因为参数如此重要，如果你在定义构造器时没有提供参数的外部名字，Swift 会为构造器的每个参数自动生成一个跟内部名字相同的外部名。

以下例子中定义了一个结构体 Color ，它包含了三个常量：red 、 green 和 blue 。这些属性可以存储 0.0 到 1.0 之间的值，用来指示颜色中红、绿、蓝成分的含量。

Color 提供了一个构造器，其中包含三个 Double 类型的构造参数。Color 也可以提供第二个构造器，它只包含名为 white 的 Double 类型的参数，它被用于给上述三个构造参数赋予同样的值。

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red    = red
        self.green   = green
        self.blue    = blue
    }
    init(white: Double) {
        red    = white
        green   = white
        blue    = white
    }
}
```

两种构造器都能用于创建一个新的 Color 实例，你需要为构造器每个外部参数传值：

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
let halfGray = Color(white: 0.5)
```

注意，如果不通过外部参数名字传值，你是没法调用这个构造器的。只要构造器定义了某个外部参数名，你就必须使用它，忽略它将导致编译错误：

```
let veryGreen = Color(0.0, 1.0, 0.0)
// 报编译时错误，需要外部名称
```

### 不带外部名的构造器参数

如果你不希望为构造器的某个参数提供外部名字，你可以使用下划线( \_ )来显式描述它的外部名，以此重写上面所说的默认行为。

下面是之前 Celsius 例子的扩展，跟之前相比添加了一个带有 Double 类型参数的构造器，其外部名用 \_ 代替：

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
    init(_ celsius: Double){
        temperatureInCelsius = celsius
    }
}
let bodyTemperature = Celsius(37.0)
// bodyTemperature.temperatureInCelsius 为 37.0
```

调用 Celsius(37.0) 意图明确，不需要外部参数名称。因此适合使用 init(\_ celsius: Double) 这样的构造器，从而可以通过提供 Double 类型的参数值调用构造器，而不需要加上外部名。

### 可选属性类型

如果你定制的类型包含一个逻辑上允许取值为空的存储型属性——无论是因为它无法在初始化时赋值，还是因为它在之后某个时间点可以赋值为空——你都需要将它定义为 可选类型（optional type）。可选类型的属性将自动初始化为 nil，表示这个属性是有意在初始化时设置为空的。

下面例子中定义了类 SurveyQuestion，它包含一个可选字符串属性 response：

```
class SurveyQuestion {
    var text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}
let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")
cheeseQuestion.ask()
// 输出 "Do you like cheese?"
cheeseQuestion.response = "Yes, I do like cheese."
```



构造器代理的实现规则和形式在值类型和类类型中有所不同。值类型（结构体和枚举类型）不支持继承，所以构造器代理的实现相对简单，因为它们只能代理给自己的其它构造器。类类型中，它可以继承自其它类（请参考继承（./13\_Inheritance.html）），这意味着类有责任保证其所有继承的存储型属性在构造时也能正确的初始化。这些责任将在后续章节类的继承和构造过程中介绍。

对于值类型，你可以使用 `self.init` 在自定义的构造器中引用相同类型中的其它构造器。并且你只能在构造器内部调用 `self.init`。

如果你为某个值类型定义了一个自定义的构造器，你将无法访问到默认构造器（如果是结构体，还将无法访问逐一成员构造器）。这种限制可以防止你为值类型增加了一个额外的且十分复杂的构造器之后，仍然有人错误的使用自动生成的构造器

注意

假如你希望默认构造器、逐一成员构造器以及你自己的自定义构造器都能用来创建实例，可以将自定义的构造器写到扩展（`extension`）中，而不是写在值类型的原始定义中。想查看更多内容，请查看扩展（./21\_Extensions.html）章节。

下面例子将定义一个结构体 `Rect`，用来代表几何矩形。这个例子需要两个辅助的结构体 `Size` 和 `Point`，它们各自为其所有的属性提供了初始值 `0.0`。

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
```

你可以通过以下三种方式为 `Rect` 创建实例——使用被初始化为默认值的 `origin` 和 `size` 属性来初始化；提供指定的 `origin` 和 `size` 实例来初始化；提供指定的 `center` 和 `size` 来初始化。在下面 `Rect` 结构体定义中，我们为这三种方式提供了三个自定义的构造器：

```
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

第一个 `Rect` 构造器 `init()`，在功能上跟没有自定义构造器时自动获得的默认构造器是一样的。这个构造器是一个空函数，使用一对大括号 `{}` 来表示，它没有执行任何构造过程。调用这个构造器将返回一个 `Rect` 实例，它的 `origin` 和 `size` 属性都使用定义时的默认值 `Point(x: 0.0, y: 0.0)` 和 `Size(width: 0.0, height: 0.0)`：

```
let basicRect = Rect()
// basicRect 的 origin 是 (0.0, 0.0), size 是 (0.0, 0.0)
```

第二个 `Rect` 构造器 `init(origin:size:)`，在功能上跟结构体在没有自定义构造器时获得的逐一成员构造器是一样的。这个构造器只是简单地将 `origin` 和 `size` 的参数值赋给对应的存储型属性：

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
// originRect 的 origin 是 (2.0, 2.0), size 是 (5.0, 5.0)
```

第三个 `Rect` 构造器 `init(center:size:)` 稍微复杂一点。它先通过 `center` 和 `size` 的值计算出 `origin` 的坐标，然后再调用（或者说代理给）`init(origin:size:)` 构造器来将新的 `origin` 和 `size` 值赋值到对应的属性中：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect 的 origin 是 (2.5, 2.5), size 是 (3.0, 3.0)
```

构造器 `init(center:size:)` 可以直接将 `origin` 和 `size` 的新值赋值到对应的属性中。然而，利用恰好提供了相关功能的现有构造器会更为方便，构造器 `init(center:size:)` 的意图也会更加清晰。

Wiki > <sup>注意</sup> 移动开发 > iOS > The Swift Programming Language 中文版

如果你想用另外一种不需要自己定义 `init()` 和 `init(origin:size:)` 的方式来实现这个例子，请参考扩展 (`./21_Extensions.html`)。

(http://

(http://

## 类的继承和构造过程

类里面的所有存储型属性——包括所有继承自父类的属性——都必须在构造过程中设置初始值。

Swift 为类类型提供了两种构造器来确保实例中所有存储型属性都能获得初始值，它们分别是指定构造器和便利构造器。

## 指定构造器和便利构造器

*指定构造器*（`designated initializers`）是类中最主要的构造器。一个指定构造器将初始化类中提供的所有属性，并根据父类链往上调用父类的构造器来实现父类的初始化。

每一个类都必须拥有至少一个指定构造器。在某些情况下，许多类通过继承了父类中的指定构造器而满足了这个条件。具体内容请参考后续章节构造器的自动继承。

*便利构造器*（`convenience initializers`）是类中比较次要的、辅助型的构造器。你可以定义便利构造器来调用同一个类中的指定构造器，并为其参数提供默认值。你也可以定义便利构造器来创建一个特殊用途或特定输入值的实例。

你应当只在必要的时候为类提供便利构造器，比方说某种情况下通过使用便利构造器来快捷调用某个指定构造器，能够节省更多开发时间并让类的构造过程更清晰明了。

## 指定构造器和便利构造器的语法

类的指定构造器的写法跟值类型简单构造器一样：

```
init(parameters) {
    statements
}
```

便利构造器也采用相同样式的写法，但需要在 `init` 关键字之前放置 `convenience` 关键字，并使用空格将它们俩分开：

```
convenience init(parameters) {
    statements
}
```

## 类的构造器代理规则

为了简化指定构造器和便利构造器之间的调用关系，Swift 采用以下三条规则来限制构造器之间的代理调用：

### 规则 1

指定构造器必须调用其直接父类的指定构造器。

### 规则 2

便利构造器必须调用同一类中定义的其它构造器。

### 规则 3

便利构造器必须最终导致一个指定构造器被调用。

一个更方便记忆的方法是：

- 指定构造器必须总是向上代理
- 便利构造器必须总是横向代理

这些规则可以通过下面图例来说明：



关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01\_The\_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02\_Basic\_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03\_Strings\_and\_Characters.html)

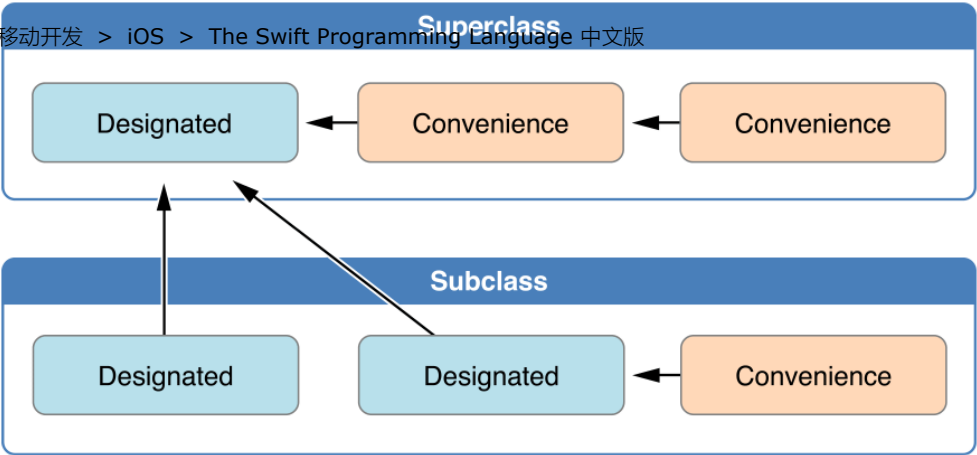
集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04\_Collection\_Types.html)

Via 由 [ 极客学院 Wiki

(http://wiki.jikexueyuan.com) ]

提供

Wiki > 移动开发 > iOS > The Swift Programming Language 中文版



如图所示，父类中包含一个指定构造器和两个便利构造器。其中一个便利构造器调用了另外一个便利构造器，而后者又调用了唯一的指定构造器。这满足了上面提到的规则 2 和 3。这个父类没有自己的父类，所以规则 1 没有用到。

子类中包含两个指定构造器和一个便利构造器。便利构造器必须调用两个指定构造器中的任意一个，因为它只能调用同一个类里的其他构造器。这满足了上面提到的规则 2 和 3。而两个指定构造器必须调用父类中唯一的指定构造器，这满足了规则 1。

注意

这些规则不会影响类的实例如何创建。任何上图中展示的构造器都可以用来创建完全初始化的实例。这些规则只影响类定义如何实现。

下面图例中展示了一种涉及四个类的更复杂的类层级结构。它演示了指定构造器是如何在类层级中充当“管道”的作用，在类的构造器链上简化了类之间的相互关系。

### 两段式构造过程

Swift 中类的构造过程包含两个阶段。第一个阶段，每个存储型属性被引入它们的类指定一个初始值。当每个存储型属性的初始值被确定后，第二阶段开始，它给每个类一次机会，在新实例准备使用之前进一步定制它们的存储型属性。

两段式构造过程的使用让构造过程更安全，同时在整个类层级结构中给予了每个类完全的灵活性。两段式构造过程可以防止属性值在初始化之前被访问，也可以防止属性被另外一个构造器意外地赋予不同的值。

注意

Swift 的两段式构造过程跟 Objective-C 中的构造过程类似。最主要的区别在于阶段 1，Objective-C 给每一个属性赋值 0 或空值（比如说 0 或 nil）。Swift 的构造流程则更加灵活，它允许你设置定制的初始值，并自如应对某些属性不能以 0 或 nil 作为合法默认值的情况。

Swift 编译器将执行 4 种有效的安全检查，以确保两段式构造过程能不出错地完成：

#### 安全检查 1

指定构造器必须保证它所在类引入的所有属性都必须先初始化完成，之后才能将其它构造任务向上代理给父类中的构造器。

如上所述，一个对象的内存只有在其所有存储型属性确定之后才能完全初始化。为了满足这一规则，指定构造器必须保证它所在类引入的属性在它往上代理之前先完成初始化。

#### 安全检查 2

指定构造器必须先向上代理调用父类构造器，然后再为继承的属性设置新值。如果没这么做，指定构造器赋予的新值将被父类中的构造器所覆盖。

#### 安全检查 3

便利构造器必须先代理调用同一类中的其它构造器，然后再为任意属性赋新值。如果没这么做，便利构造器赋予的新值将被同一类中其它指定构造器所覆盖。

#### 安全检查 4

构造器在第一阶段构造完成之前，不能调用任何实例方法，不能读取任何实例属性的值，不能引用 self 作为一个值。

类实例在第一阶段结束以前并不是完全有效的。只有第一阶段完成后，该实例才会成为有效实例，才能访问属性和调用方法。

以下是两段式构造过程中基于上述安全检查的构造流程展示：





### 构造器的自动继承

如上所述，子类在默认情况下不会继承父类的构造器。但是如果满足特定条件，父类构造器是可以被自动继承的。在实践中，这意味着对于许多常见场景你不必重写父类的构造器，并且可以在安全的情况下以最小的代价继承父类的构造器。

假设你为子类中引入的所有新属性都提供了默认值，以下 2 个规则适用：

#### 规则 1

如果子类没有定义任何指定构造器，它将自动继承所有父类的指定构造器。

#### 规则 2

如果子类提供了所有父类指定构造器的实现——无论是通过规则 1 继承过来的，还是提供了自定义实现——它将自动继承所有父类的便利构造器。

即使你在子类中添加了更多的便利构造器，这两条规则仍然适用。

注意

对于规则 2，子类可以将父类的指定构造器实现为便利构造器。

### 指定构造器和便利构造器实践

接下来的例子将在实践中展示指定构造器、便利构造器以及构造器的自动继承。这个例子定义了包含三个类 Food 、 RecipeIngredient 以及 ShoppingListItem 的类层次结构，并将演示它们的构造器是如何相互作用的。

类层次中的基类是 Food ，它是一个简单的用来封装食物名字的类。Food 类引入了一个叫做 name 的 String 类型的属性，并且提供了两个构造器来创建 Food 实例：

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

下图中展示了 Food 的构造器链：

关于 (<http://wiki.jikexueyuan.com/project/swift/>)

欢迎使用 Swift (<http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html>)

Swift 教程 (<http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html>)

基础部分 ([http://wiki.jikexueyuan.com/project/swift/chapter2/01\\_The\\_Basics.html](http://wiki.jikexueyuan.com/project/swift/chapter2/01_The_Basics.html))

基本运算符 ([http://wiki.jikexueyuan.com/project/swift/chapter2/02\\_Basic\\_Operators.html](http://wiki.jikexueyuan.com/project/swift/chapter2/02_Basic_Operators.html))

字符串和字符 ([http://wiki.jikexueyuan.com/project/swift/chapter2/03\\_Strings\\_and\\_Characters.html](http://wiki.jikexueyuan.com/project/swift/chapter2/03_Strings_and_Characters.html))

集合类型 ([http://wiki.jikexueyuan.com/project/swift/chapter2/04\\_Collection\\_Types.html](http://wiki.jikexueyuan.com/project/swift/chapter2/04_Collection_Types.html))

Via 由 [ 极客学院 Wiki

(<http://wiki.jikexueyuan.com>) ]

提供

Wiki > 移动开发 > iOS > The Swift Programming Language 中文版  
类类型没有默认的逐一成员构造器，所以 Food 类提供了一个接受单一参数 name 的指定构造器。这个构造器可以使用一个特定的名字来创建新的 Food 实例：

(<http://>

(<http://>

```
let namedMeat = Food(name: "Bacon")
// namedMeat 的名字是 "Bacon"
```

Food 类中的构造器 `init(name: String)` 被定义为一个指定构造器，因为它能确保 Food 实例的所有存储型属性都被初始化。Food 类没有父类，所以 `init(name: String)` 构造器不需要调用 `super.init()` 来完成构造过程。

Food 类同样提供了一个没有参数的便利构造器 `init()`。这个 `init()` 构造器为新食物提供了一个默认的占位名字，通过横向代理到指定构造器 `init(name: String)` 并给参数 name 传值 [Unnamed] 来实现：

```
let mysteryMeat = Food()
// mysteryMeat 的名字是 [Unnamed]
```

类层级中的第二个类是 Food 的子类 `RecipeIngredient`。`RecipeIngredient` 类构建了食谱中的一味调味料。它引入了 `Int` 类型的属性 `quantity`（以及从 Food 继承过来的 `name` 属性），并且定义了两个构造器来创建 `RecipeIngredient` 实例：

```
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

下图中展示了 `RecipeIngredient` 类的构造器链：

`RecipeIngredient` 类拥有一个指定构造器 `init(name: String, quantity: Int)`，它可以用来填充 `RecipeIngredient` 实例的所有属性值。这个构造器一开始先将传入的 `quantity` 参数赋值给 `quantity` 属性，这个属性也是唯一在 `RecipeIngredient` 中新引入的属性。随后，构造器向上代理到父类 Food 的 `init(name: String)`。这个过程满足两段式构造过程中的安全检查 1。

`RecipeIngredient` 还定义了一个便利构造器 `init(name: String)`，它只通过 name 来创建 `RecipeIngredient` 的实例。这个便利构造器假设任意 `RecipeIngredient` 实例的 `quantity` 为 1，所以不需要显式指明数量即可创建出实例。这个便利构造器的定义可以更加方便和快捷地创建实例，并且避免了创建多个 `quantity` 为 1 的 `RecipeIngredient` 实例时的代码重复。这个便利构造器只是简单地横向代理到类中的指定构造器，并为 `quantity` 参数传递 1。

注意，`RecipeIngredient` 的便利构造器 `init(name: String)` 使用了跟 Food 中指定构造器 `init(name: String)` 相同的参数。由于这个便利构造器重写了父类的指定构造器 `init(name: String)`，因此必须在前面使用 `override` 修饰符（参见构造器的继承和重写）。

尽管 `RecipeIngredient` 将父类的指定构造器重写为了便利构造器，它依然提供了父类的所有指定构造器的实现。因此，`RecipeIngredient` 会自动继承父类的所有便利构造器。

在这个例子中，`RecipeIngredient` 的父类是 Food，它有一个便利构造器 `init()`。这个便利构造器会被 `RecipeIngredient` 继承。这个继承版本的 `init()` 在功能上跟 Food 提供的版本是一样的，只是它会代理到 `RecipeIngredient` 版本的 `init(name: String)` 而不是 Food 提供的版本。

所有的这三种构造器都可以用来创建新的 `RecipeIngredient` 实例：

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

类层级中第三个也是最后一个类是 `RecipeIngredient` 的子类，叫做 `ShoppingListItem`。这个类构建了购物单中出现的某一种调味料。

购物单中的每一项总是从未购买状态开始的。为了呈现这一事实，`ShoppingListItem` 引入了一个布尔类型的属性 `purchased`，它的默认值是 `false`。`ShoppingListItem` 还添加了一个计算型属性 `description`，它提供了关于 `ShoppingListItem` 实例的一些文字描述：

极客学院

jikexueyuan.com

(http://www.jikexueyuan.com)

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01\_The\_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02\_Basic\_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03\_Strings\_and\_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04\_Collection\_Types.html)

Via 由 [ 极客学院 Wiki

(http://wiki.jikexueyuan.com) ]

提供

Wiki > 移动开发 > iOS > The Swift Programming Language 中文版

```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ✓" : " ✗"
        return output
    }
}
```

注意

ShoppingListItem 没有定义构造器来为 purchased 提供初始值，因为添加到购物单的物品的初始状态总是未购买。

由于它为自己引入的所有属性都提供了默认值，并且自己没有定义任何构造器， ShoppingListItem 将自动继承所有父类中的指定构造器和便利构造器。

下图展示了这三个类的构造器链：

你可以使用全部三个继承来的构造器来创建 ShoppingListItem 的新实例：

```
var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    print(item.description)
}
// 1 x orange juice ✓
// 1 x bacon ✗
// 6 x eggs ✗
```

如上所述，例子中通过字面量方式创建了一个数组 breakfastList，它包含了三个 ShoppingListItem 实例，因此数组的类型也能被自动推导为 [ShoppingListItem]。在数组创建完之后，数组中第一个 ShoppingListItem 实例的名字从 [Unnamed] 更改为 Orange juice，并标记为已购买。打印数组中每个元素的描述显示了它们都已按照预期被赋值。

### 可失败构造器

如果一个类、结构体或枚举类型的对象，在构造过程中有可能失败，则为其定义一个可失败构造器。这里所指的“失败”是指，如给构造器传入无效的参数值，或缺少某种所需的外部资源，又或是不满足某种必要的条件等。

为了妥善处理这种构造过程中可能会失败的情况。你可以在一个类，结构体或是枚举类型的定义中，添加一个或多个可失败构造器。其语法为在 init 关键字后面添加问号( init?)。

注意

可失败构造器的参数名和参数类型，不能与其它非可失败构造器的参数名，及其参数类型相同。

可失败构造器会创建一个类型为自身类型的可选类型的对象。你通过 return nil 语句来表明可失败构造器在何种情况下应该“失败”。

注意

严格来说，构造器都不支持返回值。因为构造器本身的作用，只是为了确保对象能被正确构造。因此你只是用 return nil 表明可失败构造器构造失败，而不要用关键字 return 来表明构造成功。

下例中，定义了一个名为 Animal 的结构体，其中有一个名为 species 的 String 类型的常量属性。同时该结构体还定义了一个接受一个名为 species 的 String 类型参数的可失败构造器。这个可失败构造器检查传入的参数是否为一个空字符串。如果为空字符串，则构造失败。否则， species 属性被赋值，构造成功。

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}
```

极客学院

jikexueyuan.com

(http://www.jikexueyuan.com)

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01\_The\_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02\_Basic\_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03\_Strings\_and\_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04\_Collection\_Types.html)

Wiki 由 [ 极客学院 Wiki

(http://wiki.jikexueyuan.com) ]

提供

移动开发 > iOS > The Swift Programming Language 中文版

```
let someCreature = Animal(species: "Giraffe")
// someCreature 的类型是 Animal? 而不是 Animal

if let giraffe = someCreature {
    print("An animal was initialized with a species of \(giraffe.species)")
}
// 打印 "An animal was initialized with a species of Giraffe"
```

如果你给该可失败构造器传入一个空字符串作为其参数，则会导致构造失败：

```
let anonymousCreature = Animal(species: "")
// anonymousCreature 的类型是 Animal?, 而不是 Animal

if anonymousCreature == nil {
    print("The anonymous creature could not be initialized")
}
// 打印 "The anonymous creature could not be initialized"
```

注意

空字符串（如 ""，而不是 "Giraffe"）和一个值为 nil 的可选类型的字符串是两个完全不同的概念。上例中的空字符串（""）其实是一个有效的，非可选类型的字符串。这里我们之所以让 Animal 的可失败构造器构造失败，只是因为对于 Animal 这个类的 species 属性来说，它更适合有一个具体的值，而不是空字符串。

### 枚举类型的可失败构造器

你可以通过一个带一个或多个参数的可失败构造器来获取枚举类型中特定的枚举成员。如果提供的参数无法匹配任何枚举成员，则构造失败。

下例中，定义了一个名为 TemperatureUnit 的枚举类型。其中包含了三个可能的枚举成员( Kelvin， Celsius，和 Fahrenheit )，以及一个根据 Character 值找出所对应的枚举成员的可失败构造器：

```
enum TemperatureUnit {
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
            case "K":
                self = .Kelvin
            case "C":
                self = .Celsius
            case "F":
                self = .Fahrenheit
            default:
                return nil
        }
    }
}
```

你可以利用该可失败构造器在三个枚举成员中获取一个相匹配的枚举成员，当参数的值不能与任何枚举成员相匹配时，则构造失败：

```
let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
// 打印 "This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}
// 打印 "This is not a defined temperature unit, so initialization failed."
```

### 带原始值的枚举类型的可失败构造器

带原始值的枚举类型会自带一个可失败构造器 init?(rawValue:)，该可失败构造器有一个名为 rawValue 的参数，其类型和枚举类型的原始值类型一致，如果该参数的值能够和某个枚举成员的原始值匹配，则该构造器会构造相应的枚举成员，否则构造失败。

因此上面的 TemperatureUnit 的例子可以重写为：

http://wiki.jikexueyuan.com/project/swift/chapter2/14\_Initialization.html

12/18

Wiki > 移动开发 => iOS => The Swift Programming Language 中文版

```
enum TemperatureUnit: Character {
    case fahrenheit = "F", celsius = "C"
}

let fahrenheitUnit = TemperatureUnit(rawValue: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
// 打印 "This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(rawValue: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}
// 打印 "This is not a defined temperature unit, so initialization failed."
```

### 构造失败的传递

类，结构体，枚举的可失败构造器可以横向代理到类型中的其他可失败构造器。类似的，子类的可失败构造器也能向上代理到父类的可失败构造器。

无论是向上代理还是横向代理，如果你代理到的其他可失败构造器触发构造失败，整个构造过程将立即终止，接下来的任何构造代码不会再被执行。

注意

可失败构造器也可以代理到其它的非可失败构造器。通过这种方式，你可以增加一个可能的失败状态到现有的构造过程中。

下面这个例子，定义了一个名为 `CartItem` 的 `Product` 类的子类。这个类建立了一个在线购物车中的物品的模型，它有一个名为 `quantity` 的常量存储属性，并确保该属性的值至少为 1：

```
class Product {
    let name: String
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}

class CartItem: Product {
    let quantity: Int
    init?(name: String, quantity: Int) {
        if quantity < 1 { return nil }
        self.quantity = quantity
        super.init(name: name)
    }
}
```

`CartItem` 可失败构造器首先验证接收的 `quantity` 值是否大于等于 1。倘若 `quantity` 值无效，则立即终止整个构造过程，返回失败结果，且不再执行余下代码。同样地，`Product` 的可失败构造器首先检查 `name` 值，假如 `name` 值为空字符串，则构造器立即执行失败。

如果你通过传入一个非空字符串 `name` 以及一个值大于等于 1 的 `quantity` 来创建一个 `CartItem` 实例，那么构造方法能够成功被执行：

```
if let twoSocks = CartItem(name: "sock", quantity: 2) {
    print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")
}
// 打印 "Item: sock, quantity: 2"
```

倘若你以一个值为 0 的 `quantity` 来创建一个 `CartItem` 实例，那么将导致 `CartItem` 构造器失败：

```
if let zeroShirts = CartItem(name: "shirt", quantity: 0) {
    print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")
} else {
    print("Unable to initialize zero shirts")
}
// 打印 "Unable to initialize zero shirts"
```

同样地，如果你尝试传入一个值为空字符串的 `name` 来创建一个 `CartItem` 实例，那么将导致父类 `Product` 的构造过程失败：

Wiki >

```
if let oneUnnamed = CartItem(name: "", quantity: 1) {
    print("Item: \((oneUnnamed.name, quantity: oneUnnamed.quantity)")
} else {
    print("Unable to initialize one unnamed product")
}
// 打印 "Unable to initialize one unnamed product"
```

(<http://>

如同其它的构造器，你可以在子类中重写父类的可失败构造器。或者你也可以用子类的非可失败构造器重写一个父类的可失败构造器。这使你定义一个不会构造失败子类，即使父类的构造器允许构造失败。

注意，当你用子类的非可失败构造器重写父类的可失败构造器时，向上代理到父类的可失败构造器的唯一方式是对父类的可失败构造器的返回值进行强制解包。

注意

你可以用非可失败构造器重写可失败构造器，但反过来却不行。

下例定义了一个名为 Document 的类，name 属性的值必须为一个非空字符串或 nil，但不能是一个空字符串：

```
class Document {
    var name: String?
    // 该构造器创建了一个 name 属性的值为 nil 的 document 实例
    init() {}
    // 该构造器创建了一个 name 属性的值为非空字符串的 document 实例
    init?(name: String) {
        self.name = name
        if name.isEmpty { return nil }
    }
}
```

下面这个例子，定义了一个 `Document` 类的子类 `AutomaticallyNamedDocument`。这个子类重写了父类的两个指定构造器，确保了无论是使用 `init()` 构造器，还是使用 `init(name:)` 构造器并为参数传递空字符串，生成的实例中的 `name` 属性总有初始值 `"[Untitled]"`：

```
class AutomaticallyNamedDocument: Document {
    override init() {
        super.init()
        self.name = "[Untitled]"
    }
    override init(name: String) {
        super.init()
        if name.isEmpty {
            self.name = "[Untitled]"
        } else {
            self.name = name
        }
    }
}
```

`AutomaticallyNamedDocument` 用一个非可失败构造器 `init(name:)` 重写了父类的可失败构造器 `init?(name:)`。因为子类用另一种方式处理了空字符串的情况，所以不再需要一个可失败构造器，因此子类用一个非可失败构造器代替了父类的可失败构造器。

你可以在子类的非可失败构造器中使用强制解包来调用父类的可失败构造器。比如，下面的 `UntitledDocument` 子类的 `name` 属性的值总是 `"[Untitled]"`，它在构造过程中使用了父类的可失败构造器 `init?(name:)`：

```
class UntitledDocument: Document {
    override init() {
        super.init(name: "[Untitled]")!
    }
}
```

在这个例子中，如果在调用父类的可失败构造器 `init?(name:)` 时传入的是空字符串，那么强制解包操作会引发运行时错误。不过，因为这里是通过非空的字符串常量来调用它，所以并不会发生运行时错误。

通常来说我们通过在 `init` 关键字后添加问号的方式（`init?`）来定义一个可失败构造器，但你也可以通过在 `init` 后面添加惊叹号的方式来定义一个可失败构造器（`init!`），该可失败构造器将会构建一个对应类型的隐式解包可选类型的对象。





关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01\_The\_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02\_Basic\_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03\_Strings\_and\_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04\_Collection\_Types.html)

Via 由 [ 极客学院 Wiki

(http://wiki.jikexueyuan.com) ]

提供

Wiki > 移动开发 > iOS > The Swift Programming Language 中文版

```
struct Checkerboard {
    let boardColors: [Bool]
    var temporaryBoard = [Bool]()
    var isBlack = false
    for i in 1...8 {
        for j in 1...8 {
            temporaryBoard.append(isBlack)
            isBlack = !isBlack
        }
        isBlack = !isBlack
    }
    return temporaryBoard
}()

func squareIsBlackAtRow(row: Int, column: Int) -> Bool {
    return boardColors[(row * 8) + column]
}
```


每当一个新的 Checkerboard 实例被创建时，赋值闭包会被执行， boardColors 的默认值会被计算出来并返回。上面例子中描述的闭包将计算出棋盘中每个格子对应的颜色，并将这些值保存到一个临时数组 temporaryBoard 中，最后在构建完成时将此数组作为闭包返回值返回。这个返回的数组会保存到 boardColors 中，并可以通过工具函数 squareIsBlackAtRow 来查询：

```
let board = Checkerboard()
print(board.squareIsBlackAtRow(0, column: 1))
// 打印 "true"
print(board.squareIsBlackAtRow(7, column: 7))
// 打印 "false"
```

上一篇：继承 (/project/swift/chapter2/13\_Inheritance.html)

下一篇：析构过程 (/project/swift/chapter2/15\_Deinitialization.html)

被顶起来的评论



dong (http://weibo.com/wenbokenet)

在可失败构造中，确实是有问题的，在变量常量隐式可选，返回nil和赋值的先后，是有问题的，虽然说有规则是在类的可失败构造中返回nil前，所有的存储属性要被初始化，隐式可选本身就是存在默认值nil，所以是满足的，在描述中，说的就是这个意思，可以在赋值前发现参数不合法返回nil，可是代码却是先给name赋值，意思是属性要初始化完全后才可以检查参数，和后一段描述不符，前后矛盾。其实变量隐式可选在类的初始化方法中是不需要给值的，符合后一段描述的，无奈这里是个常量的，必须给它赋值，编译器才认为是初始化完全的，进而才可以进行参数判断，问题就出在这个常量上了，不首先给它赋值，不合乎类的可失败构造规则，而如果是变量，就可以正常构造，默认值是nil，编译器认为是初始化完全的，可以先判断参数的合法性，返回nil，合乎规则，可是我觉得问题就出在这个规则，为什么要初始化完全后，才可以进行参数的合法性检查，逻辑上就不太对，检查到不合法的参数就构造失败，就不需要为属性赋值了，都构造失败了，就不需要这里的内存了。据我所查到的资料，这里是有bug的，在未初始化完全的情况下，应该是可以返回nil的，无奈这里说这是个规则，无限费解。🤔

1月4日    回复    顶(1)    转发

21条评论

4条新浪微博

最新 最早 最热

- 

fangry0823

2.2已经翻译出来了，真是给力。十分感谢翻译组的成员。

4月6日    回复    顶    转发
- 

heiheihei

必要构造器那里，有个注意：“如果子类继承的构造器能满足必要构造器的要求，则无须在子类中显式提供必要构造器的实现。”，请问这的“满足必要构造器的要求”，是指什么要求？

3月3日    回复    顶    转发
- 

swift


回复 enzou: xcode7 不会报错 你试试

2月29日    回复    顶    转发
- 

从今以后

回复 dong: 这个貌似要在3.0修复了

2月15日    回复    顶    转发



dong (http://weibo.com/wenbokenet)


在可失败构造中，确实是有问题的，在变量常量隐式可选，返回nil和赋值的先后，是有问题的，虽然说有规则是在类的可失败构造中返回nil前，所有的存储属性要被初始化，隐式可选本身就是存在默认值nil，所以是满足的，在描述中，说的就是这个意思，可以在赋值前发现参数不合法返回nil，可是代码却是先给name赋值，意思是属性要初始化完全后才可以检查参数，和后一段描述不符，前后矛盾。其实变量隐式可选在类的初始化方法中是不需要给值的，符合后一段描述的，无奈这里是个常量

http://wiki.jikexueyuan.com/project/swift/chapter2/14\_Initialization.html

16/18


的，必须给它赋值，编译器才认为是初始化完全的，进而才可以进行参数判断，问题就出在这个常量上了，不首先给它赋值，不合乎类的可失败构造规则，而如果是变量，就可以正常构造。默认值是nil，编译器认为是初始化完全的，可以先判断参数的合法性，返回nil，合乎规则，可是我觉得问题就出在这个规则，为什么要初始化完全后，才可以进行参数的合法性检查，逻辑上就不太对，检查到不合法的参数就构造失败，就不需要为属性赋值了，都构造失败了，就不需要这里的内存了。据我所查到的资料，这里是有bug的，在未初始化完全的情况下，应该是可以返回nil的，无奈这里说这是个规则，无限费解。😓

1月4日    回复    顶(1)    转发

 dong (http://weibo.com/wenbokenet)

很复杂啊，理不清楚


(http://weibo.com/wenbokenet)1月4日    回复    顶    转发

 沉沦2013

```
class Product {
    let name: String!
    init?(name: String) {
        self.name = name
        if name.isEmpty { return nil }
    }
}
```

我觉得这里并没有任何问题，因为根据安全检查1，首先要对本类新引进的存储属性初始化，然后安全检查2，才会调用父类构造器完成初始化，这样看来，可失败构造器都是在完成所有构造后，才对属性进行失败构造相应条件的构造，比如这里的条件是name.isEmpty，但这个条件的前提是name必须不是nil，所以才声明成隐式解包。如果把self.name = name 放在后面，这里的self.name仅仅是默认值nil，name.isEmpty就不能执行了。如果放在后面，name已经被构造，然后再被赋值，而name是let当然会编译失败，而放在后面如果声明称var，那么编译就会通过。let属性即使有默认值，在被构造之前是可以赋值的，一旦被构造在赋值，就会编译失败了。


2015年12月13日    回复    顶    转发



从今以后 (http://t.qq.com/congjinyih6382)

回复 enzou: Xcode 7 下无问题，应该是 Swift 2.0 的问题吧。


(http://t.qq.com/congjinyih6382)2015年11月16日    回复    顶    转发



从今以后 (http://t.qq.com/congjinyih6382)


回复 longhh1984: 这句话的意思是，调用一个便利构造器时，最后必须导致一个指定构造器被调用。

(http://t.qq.com/congjinyih6382)2015年11月16日    回复    顶    转发

 magi

the only way to delegate up to the superclass initializer is to force-unwrap the result of the failable superclass initializer. 子类的构造器将不再能向上代理父类的可失败构造器。一个非可失败的构造器永远也不能代理调用一个可失败构造器。翻译错误,应该为唯一方式调用父类的可失败构造函数是对父类的可失败构造函数强制解包。

2015年11月3日    回复    顶    转发

 (http://weibo.com/2159582061)


小名儿哈哈 (http://weibo.com/2159582061)

除上面提到的默认构造器，如果结构体对所有存储型属性提供了默认值且自身没有提供定制的构造器，它们能自动获得一个逐一成员构造器。翻译的有问题，

Structure types automatically receive a memberwise initializer if they do not define any of their own custom initializers. Unlike a default initializer, the structure receives a memberwise initializer even if it has stored properties that do not have default values.

结构体的逐一成员构造器:结构体类型将自动获得一个逐一成员构造器如果自身没有定义任何构造器。与默认构造器不同的是,就算,存储型属性没有提供默认值,结构体类型也会获得一个逐一成员构造器


2015年10月24日    回复    顶    转发

 (http://weibo.com/2159582061)

小名儿哈哈 (http://weibo.com/2159582061)

"类实例在第一阶段结束以前并不是完全有效，仅能访问属性和调用方法，一旦完成第一阶段，该实例才会声明为有效实例。" 翻译有误,"The class instance is not fully valid until the first phase ends. Properties can only be accessed, and methods can only be called, once the class instance is known to be valid at the end of the first phase." 在第一构造过程的第一阶段完成之后,属性才能够被访问,实例方法才能够被调用。

2015年10月22日    回复    顶    转发

 a


回复 锅巴GG不唠嗑: 确实，翻译有误，应该翻译为"便利构造器必须最终调用一个指定构造器"，无论是直接调用还是通过构造器链。

2015年10月22日    回复    顶    转发

 tdt

struct Celsius {I多了个I吧。

2015年10月12日    回复    顶    转发




张十三 (http://t.qq.com/charles\_zhang13)

回复 enzou: 可以在构造器里给没有初始化的常量复制的啊！不会报错啊

(http://t.qq.com/charles\_zhang13)2015年10月11日    回复    顶    转发


Wiki > 移动开发 > iOS > The Swift Programming Language 中文版



class CartItem: Product {  
let quantity: Int!  
init?(name: String, quantity: Int) {  
self.quantity = quantity  
super.init(name: name)  
if quantity < 1 { return nil }  
}  
}

当令let quantity: Int!编译器会报错，提示要改成var变量


2015年10月7日    回复    顶    转发



锅巴GG不唠嗑 (http://weibo.com/g8gg)

class SomeClassNew {  
let someProperty: String = { //SomeType is String  
// 在这个闭包中给 someProperty 创建一个默认值  
// someValue 必须和 SomeType 类型相同  
return ""//someValue  
}()}//注意闭包结尾的大括号后面接了一对空的小括号。这是用来告诉 Swift 需要立刻执行此闭包。如果你忽略了这对括号，相当于是将闭包本身作为值赋值给了属性，而不是将闭包的返回值赋值给属性。  
}


2015年10月3日    回复    顶    转发



denchn

回复 longhh1984: 便利构造器 实质是调用的本类其他构造器（可以是其他便利构造器或指定构造器），但通过构造器链迭代下去最终会以一个本类的指定构造器来结束构造器链。还有，根据安全检查3，便利构造器调用其他构造器的代码必须放在最开始，而不是最后


2015年10月1日    回复    顶    转发



Yang\_Home (http://weibo.com/iphone4china)

少了这个例子  
class UntitledDocument: Document {  
override init() {  
super.init(name: "[Untitled]")!  
}  
}

2015年9月21日    回复    顶    转发




zyg

class CartItem: Product {  
let quantity: Int!  
init?(name: String, quantity: Int) {  
self.quantity = quantity  
super.init(name: name)  
if quantity < 1 { return nil }  
}  
}

2015年8月13日    回复    顶    转发

12

社交帐号登录:    微信    微博    QQ    人人    更多»



说点什么吧...

发布

[ 极客学院 Wiki - wiki.jikexueyuan.com ] 正在使用多说 (http://duoshuo.com)

http://wiki.jikexueyuan.com/project/swift/chapter2/14\_Initialization.html

18/18